Middleware Pain? Meet iceoryx2

FOSDEM 2026

Michael Pöhnl

Jan 31, 2026

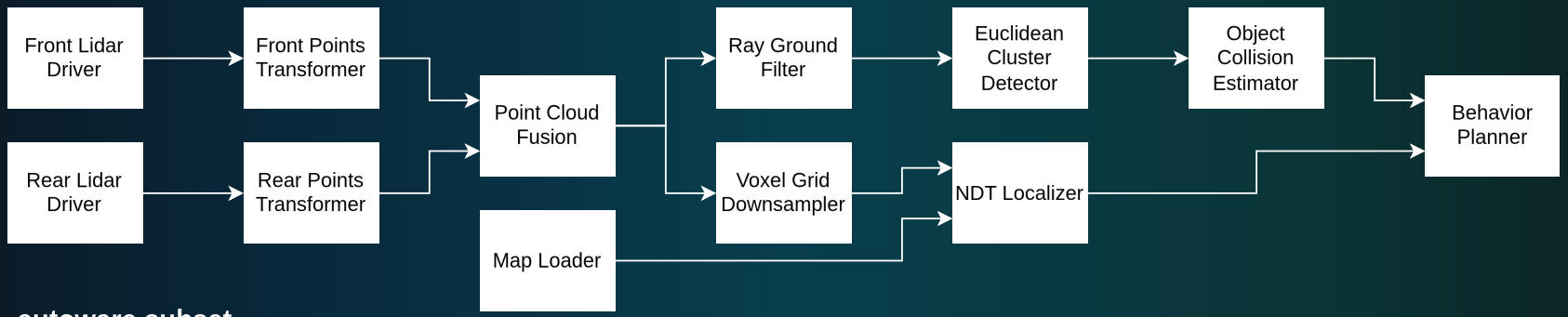
Agenda

1. Why do you even need middleware?
2. Introduction to Eclipse iceoryx™
3. The pain iceoryx2 relieves
4. iceoryx2 community insights

Why do you even need middleware?

All about DAGs

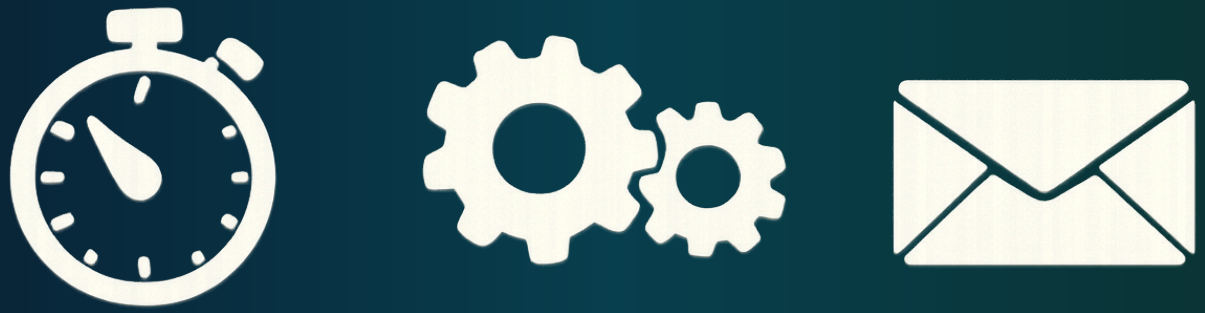
- Robotics applications are typically directed acyclic graphs (DAGs)
- Nodes are the processing steps, edges are communication links
- The whole graph follows a data-in / data-out model



autoware subset

All about communication and execution

So you must manage when execution happens and which data is used



Let’s build our robotics application

- You start with some timers, read some sensor data from network sockets
- ...hey, why not use several threads to make use of the multi-core CPU...
- ...hey, why not several processes connected via an OS IPC mechanism...
- ...this needs some glue code to orchestrate communication and execution...
- ...it shall run on different operating systems, so let’s add some #ifdefs...
- ...phew, let’s clean it up a bit with abstraction layers to make things nicer...

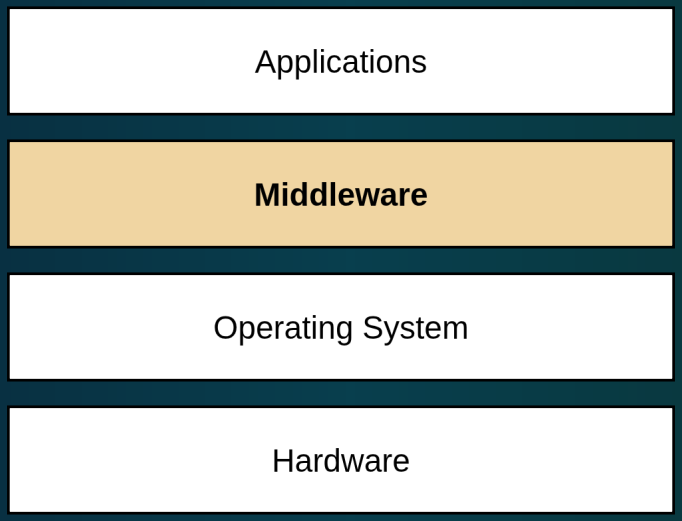
Welcome to the club

Congratulations, you are building the next middleware!

Photo by Jason Leung on Unsplash

Why open-source middleware is great

- Middleware is infrastructure software, needed across many domains
- It's not trivial: low-level stuff, multi-threaded programming, etc.
- Reinventing the wheel makes no sense if your business is not middleware
- With a robust OSS solution, teams can focus on applications and products
- ROS is one example, allowing roboticists to focus on robotics (ideally)



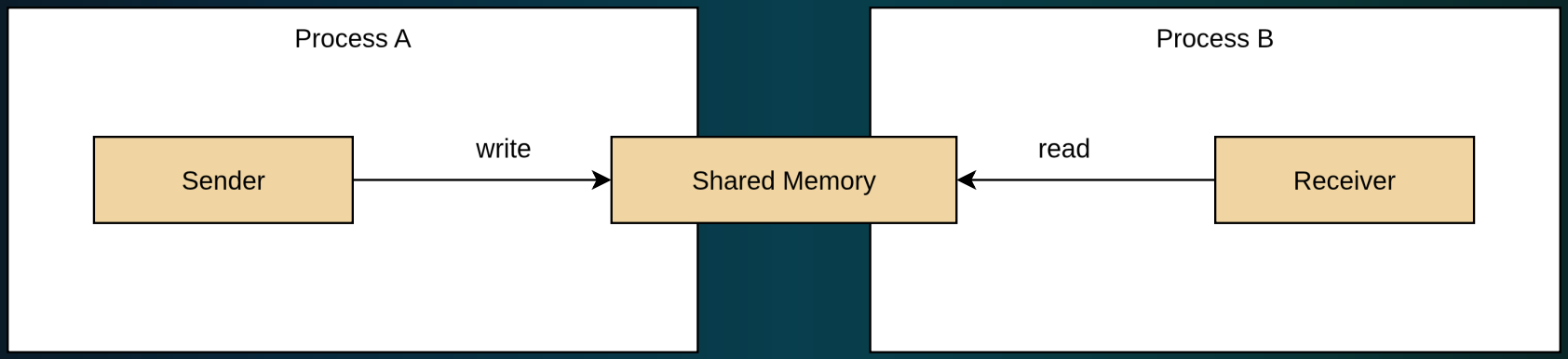
Introduction to Eclipse iceoryx™

What is Eclipse iceoryx?

- OSS project hosted by the Eclipse Foundation
- Inter-process communication (IPC) based on shared memory
- Open-source since 2019; second generation (iceoryx2) started in 2023
- Based on 70+ years of combined experience in high-performance IPC

How does it work?

- Sender writes directly into shared memory
- Receiver reads directly from shared memory
- iceoryx provides the whole communication infrastructure
- iceoryx takes care of discovery, message delivery, memory management
- This is what we call true zero-copy communication



Next Gen iceoryx2 already surpasses the first generation

- Written in Rust, runs everywhere
  - Linux, Windows, macOS, QNX, VxWorks, FreeBSD, eMCOS, bare metal
  - In addition to Rust APIs, language bindings for C, C++, Python, C#
- Huge set of messaging patterns
  - publish/subscribe, request/response, key/value storage (blackboard)
  - Event mechanism for data-driven triggering of execution
- Developed for mission-critical systems
  - No heap allocation during runtime, no blocking calls, Rust “no_std”
  - Decentralized and mixed-criticality architecture

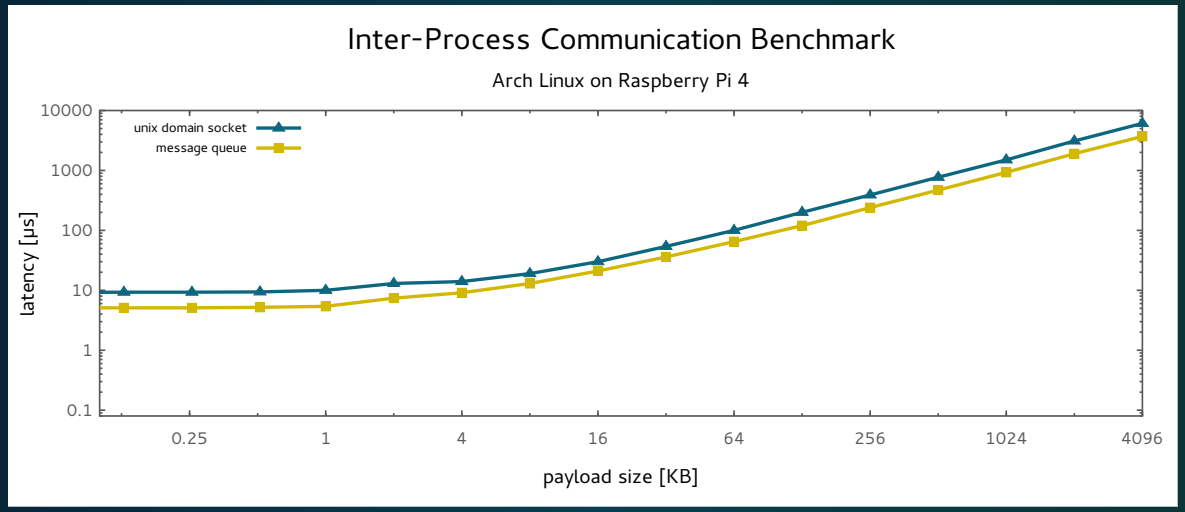
OK, an IPC middleware. What else?

The pain iceoryx2 relieves

Causes pain: copies and serialization in IPC

Why copies and serialization inside your IPC cause pain

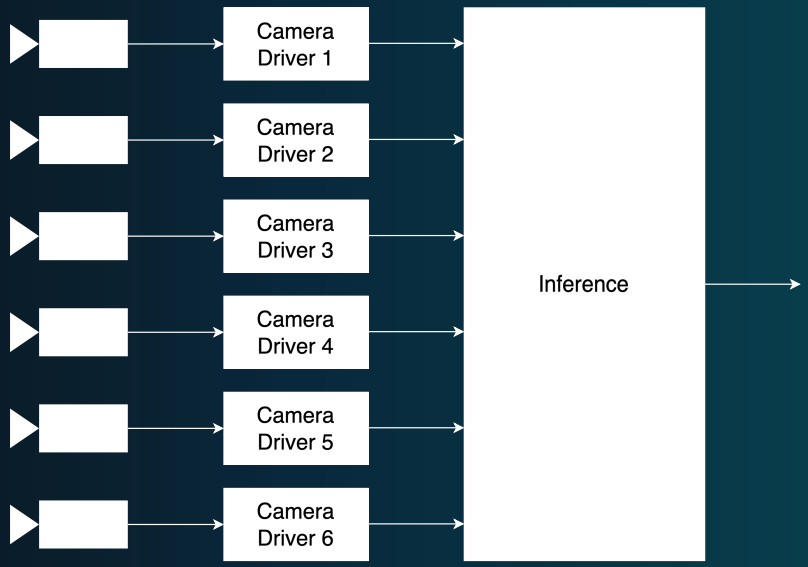
- This takes CPU cycles, the bigger the message, the more of them
- Advanced robotics systems can have tens of GB/s IPC
- Consequences
  - High CPU load
  - High latency



Causes pain: copies and serialization in IPC

Real-life pain example

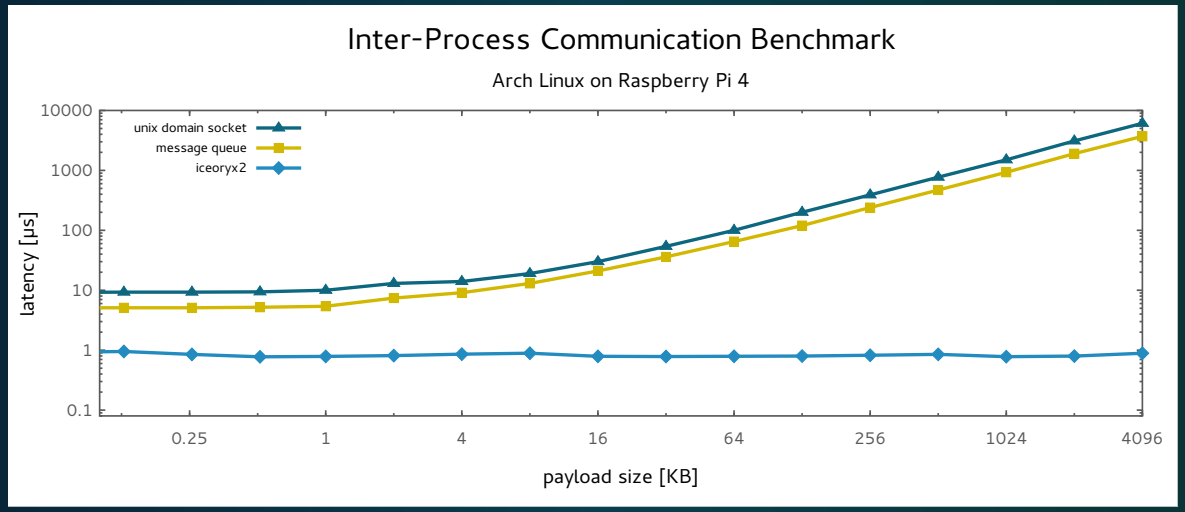
- Autonomous driving based on an end-to-end AI model
- 6 cameras × ~35 MB × 20 fps ⟶ more than 4 GB/s camera data stream
- Now think: copying on sender side, copying on receiver side, recording...



Relieves pain: iceoryx2's true zero-copy IPC

How iceoryx2 relieves the pain

- True zero-copy IPC
- Constant sub-microsecond latency, independent of message size
- More than 1000 times less latency for MB messages (think < µs, not > ms)



Relieves pain: iceoryx2's true zero-copy IPC

What it enables

- Dozens of GB/s IPC with ultra-low CPU load
- Fast reaction times from sensing to acting
- Enables use of smaller, cheaper CPUs for the same workload
- Scalability: going embedded without going crazy

Photo by Leo_Visions on Unsplash

Causes pain: threads inside the middleware

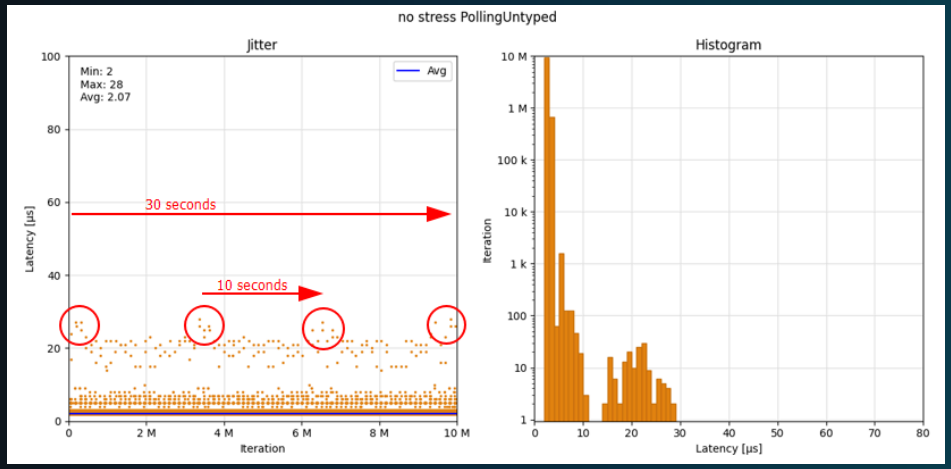
Why additional threads inside the middleware cause pain

- Background threads in the middleware for discovery, housekeeping, etc.
  - When do they wake up to do what?
  - How to configure these threads?
- Often additional middleware threads involved in the message passing
  - When will the message be delivered?
  - A context switch isn't expensive, but how about thousands per second?
- Consequences
  - Not deterministic
  - High scheduling overhead

Causes pain: threads inside the middleware

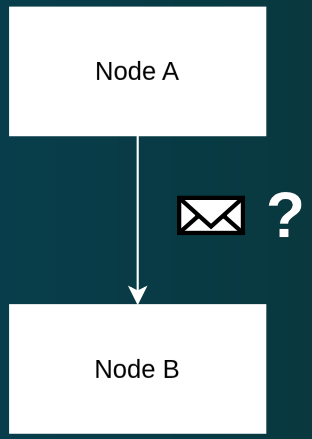
Real-life pain examples

Strange latency spikes
(Interference from middleware threads)



iceoryx classic: interference from monitoring thread

Message visibility race
(Middleware threads pass the message)




```
int main()
{

....

nodeA.run(); // sends a message

nodeB.run(); // can read the message, or?

...

}
```

Relieves pain: iceoryx2 has no threads

How iceoryx2 relieves the pain

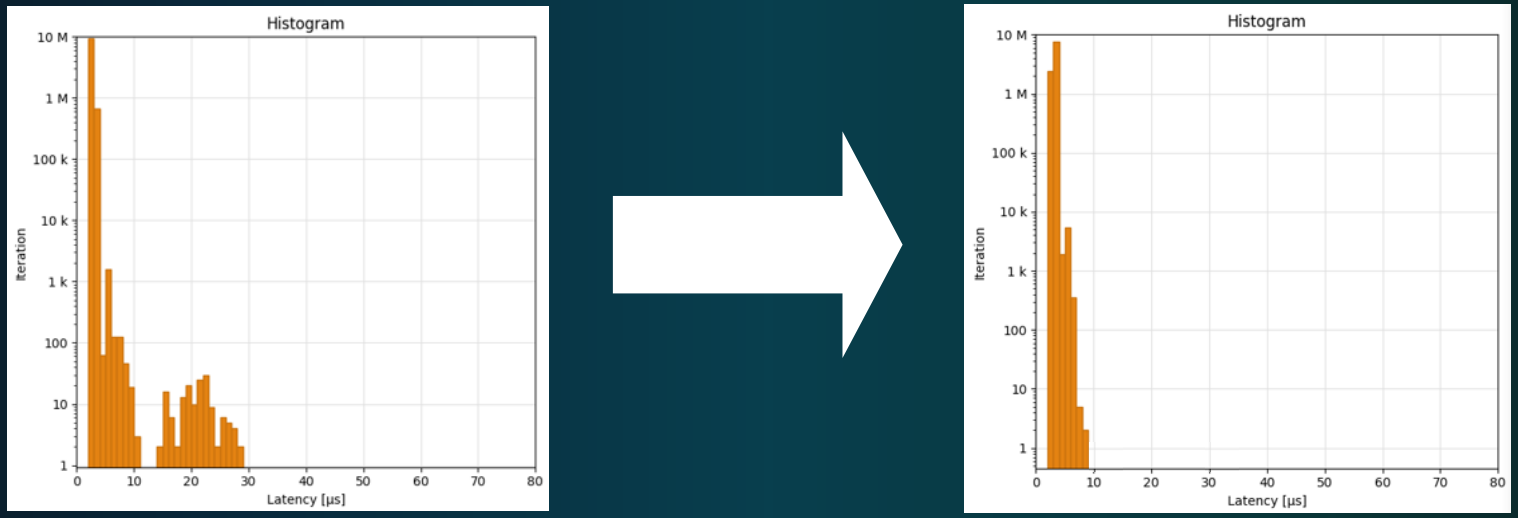
- iceoryx2 comes with no internal threads.
- Housekeeping is done when entities are created or destroyed
- No surprises behind the scenes

Photo by Keagan Henman on Unsplash

Relieves pain: iceoryx2 has no threads

What it enables

- Together with zero-copy IPC, stable low latencies with low jitter
- Deterministic behavior, no mysterious background actions
- Reliable message passing along a single thread of execution
- Lower scheduling overhead, lower CPU usage



iceoryx classic: after disabling the monitoring thread

Causes pain: improper queuing in IPC

Why improper message queuing inside IPC channels results in pain

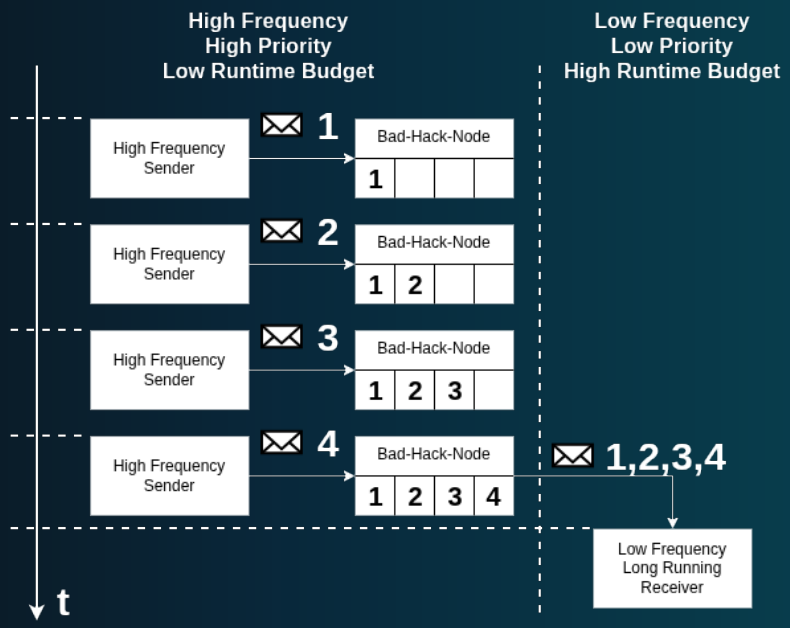
- When there is no message queuing
  - Sender may outpace receiver and overwrite unread messages
- When there is a queue but it overflows
  - Sender gets blocked or drops new messages
- When there are no historical messages
  - Startup order must be controlled to not miss messages
- Consequences
  - Backpressure from the receiver to the sender
  - Receiver is forced to keep pace with sender to not miss messages

Causes pain: improper queuing in IPC

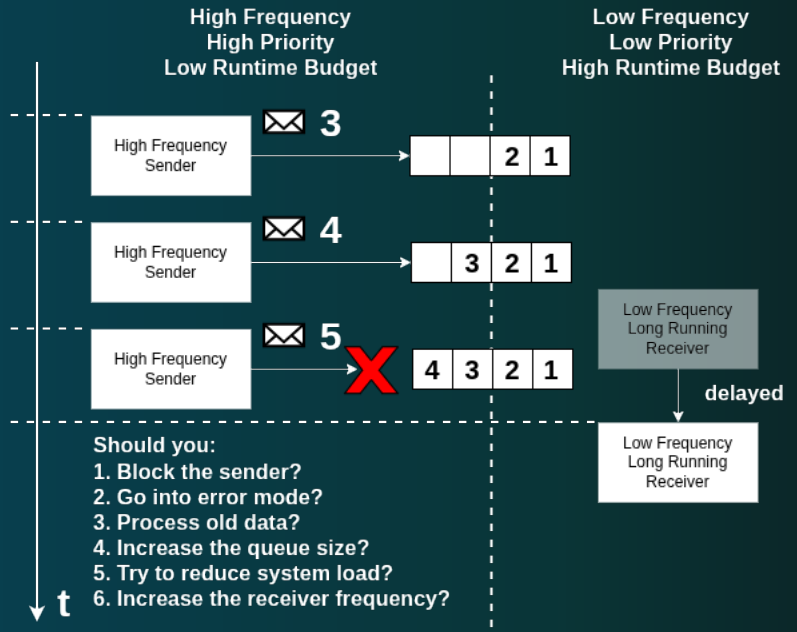
Real-life pain example

Use case: low frequency consumer needs 4 latest messages

When all you needed was a queue



When a usual queue is not enough



Relieves pain: iceoryx2's queuing and history possibilities

How iceoryx2 relieves the pain

- Per receiver queues with configurable size (N)

- Safely overflowing queues
  - Provides the latest N messages in case of overflow
  - Drops old, unread messages in favor of new ones

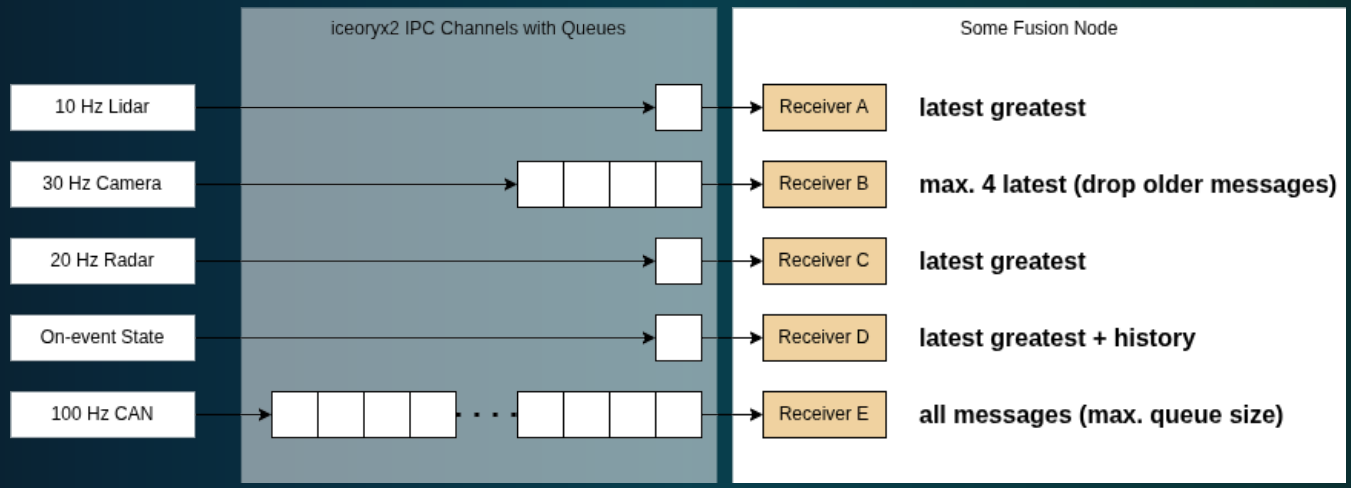
- Queue overflow behavior can be configured
  - Return an error and drop the new message
  - Block the sender, wait for the receiver
  - Overflow and drop oldest message

- Optional history on the sender side for late joining receivers

Relieves pain: iceoryx2's queuing and history possibilities

What it enables

- Decoupling of senders and receivers
- Memory efficiency by only queuing messages of interest
- Different message consumption patterns, your choice
- DDS-style message caching on higher middleware layers



Causes pain: per-message callbacks

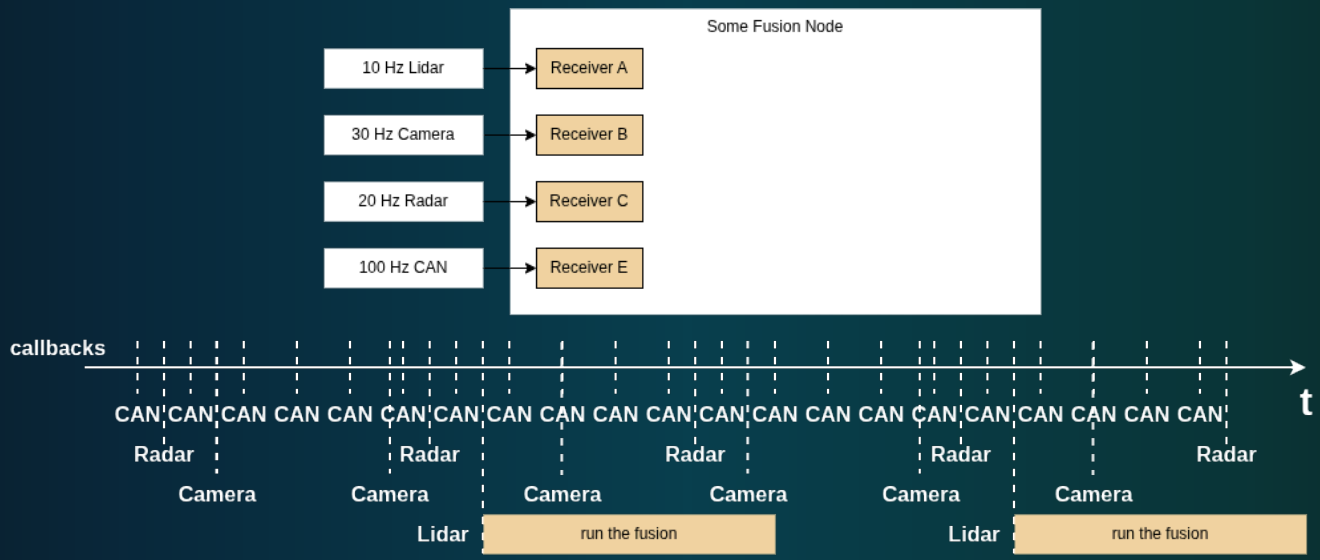
Why per-message callbacks result in pain

- This couples communication and execution
  - Business logic is forced to react on every message
- When you cannot access the receivers side by side
  - Forces you to do message caching across callbacks
- When the message only lives for the callback duration
  - Forces you to copy the message if you want a later processing
- Consequences
  - Many context switches, high scheduling overhead
  - More administrative effort on the user side

Causes pain: per-message callbacks

Real-life pain example

- Fusion node with messages arriving at different frequencies
- Algorithm collects messages and processes them simultaneously (on Lidar)
- In this simple example: 16 individual callbacks for 1 fusion run



Relieves pain: iceoryx2's decoupled event pattern

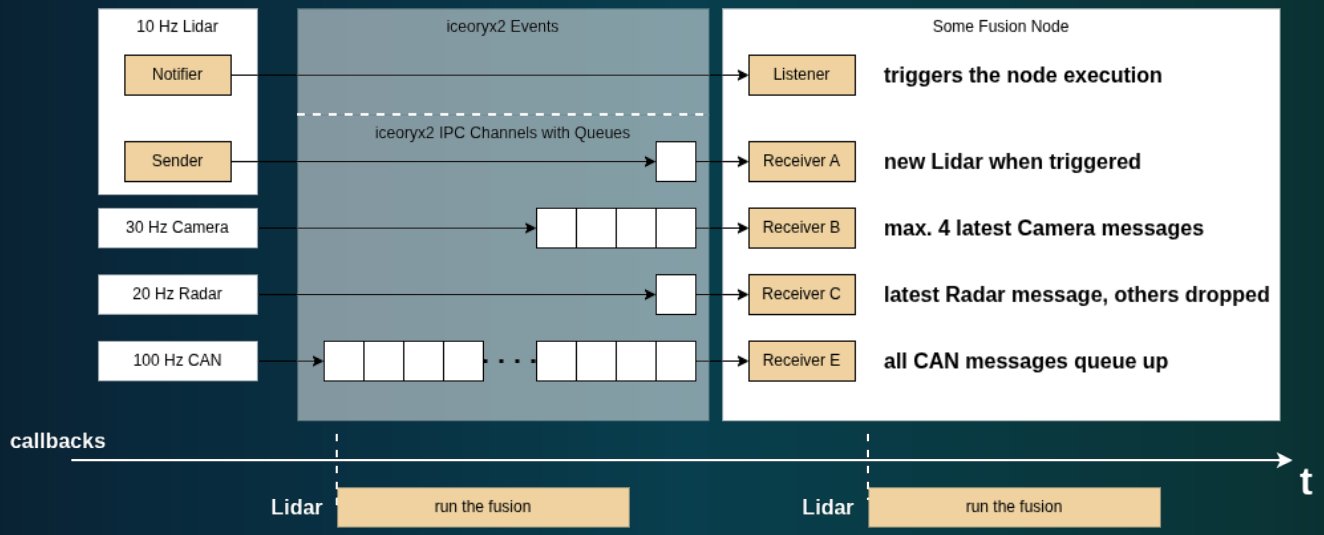
How iceoryx2 relieves the pain

- Notification with related context switching is decoupled from messaging
- Event is a separate pattern with notifiers, listeners and waitsets
  - Notifier: sends the notification on event
  - Listener: can wait on an event to occur
  - Waitset: allows to wait on many events within a single thread
- Events can easily be combined with publish/subscribe or request/response
  - e.g. a triggering publisher that always notifies when sending
- No problem to implement per-message callbacks if this is your philosophy

Relieves pain: iceoryx2's decoupled event pattern

What it enables

- Trigger on relevant messages only, efficiently queue the others
- Using iceoryx2 events for other notifications (e.g. timer events)
- iceoryx2 enables you to implement your specific execution strategy
- Less context switching, more CPU time for your applications



iceoryx2 community insights

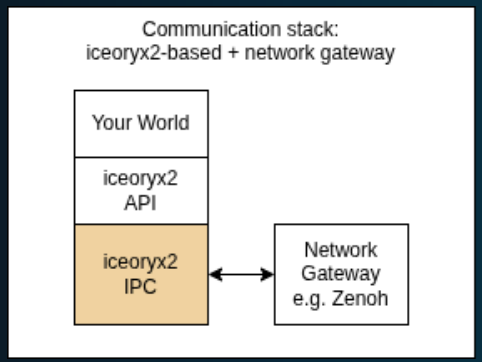
iceoryx2 adopters

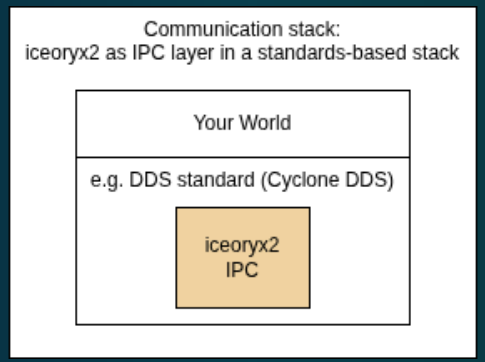
Vast community with a need for low latency and high-volume data transfer

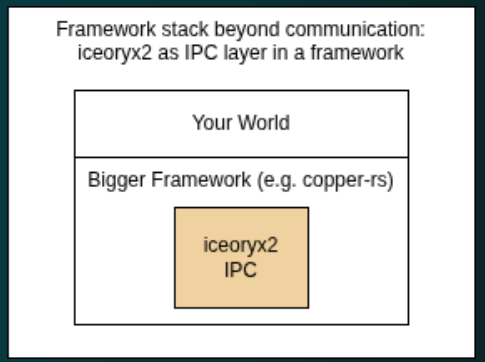
iceoryx2 in the bigger middleware context

iceoryx2 is primarily a shared memory communication technology and can

- be combined with network protocols for a full communication stack
  - e.g. with Zenoh, gRPC, MQTT, DDS, SOME/IP
- be integrated as IPC layer in a broader communication stack
  - e.g. based on a standard like DDS defining an IDL and data model
- be integrated as IPC layer in a framework going beyond communication
  - e.g. in copper-rs, Eclipse eCAL, via rmw_iceoryx2 in ROS 2







Questions?

iceoryx